(12) UK Patent Application (19) GB (11) 2 343 022 (13) A

(43) Date of A Publication 26.04.2000

(21) Application No 9822834.9

(22) Date of Filing 19.10.1998

(51) INT CL[7]
G06F 1/00

(52) UK CL (Edition R)
G4A AAP

(56) Documents Cited
EP 0875815 A2 EP 0875814 A2

(58) Field of Search
UK CL (Edition Q) G4A AAP AFL
INT CL[6] G06F 1/00 12/14
Online: WPI

(71) Applicant(s)
International Business Machines Corporation
(Incorporated in USA - New York)
Armonk, New York 10504, United States of America

(72) Inventor(s)
Alan Michael Webb

(74) Agent and/or Address for Service
P Waldner
IBM United Kingdom Limited, Intellectual Property Department, Hursley Park, WINCHESTER, Hampshire, SO21 2JN, United Kingdom

(54) Abstract Title
Encrypting of Java methods

(57) A method of processing a Java ClassFile component on a client comprising retrieving the ClassFile component from a server and checking the component for a compiled part which is indicated by a flag or attribute in the ClassFile. If a compiled part is detected further checking of the ClassFile is performed to see if the part is encrypted, again indicated by a flag or attribute. If the part is so encrypted it is then decrypted.

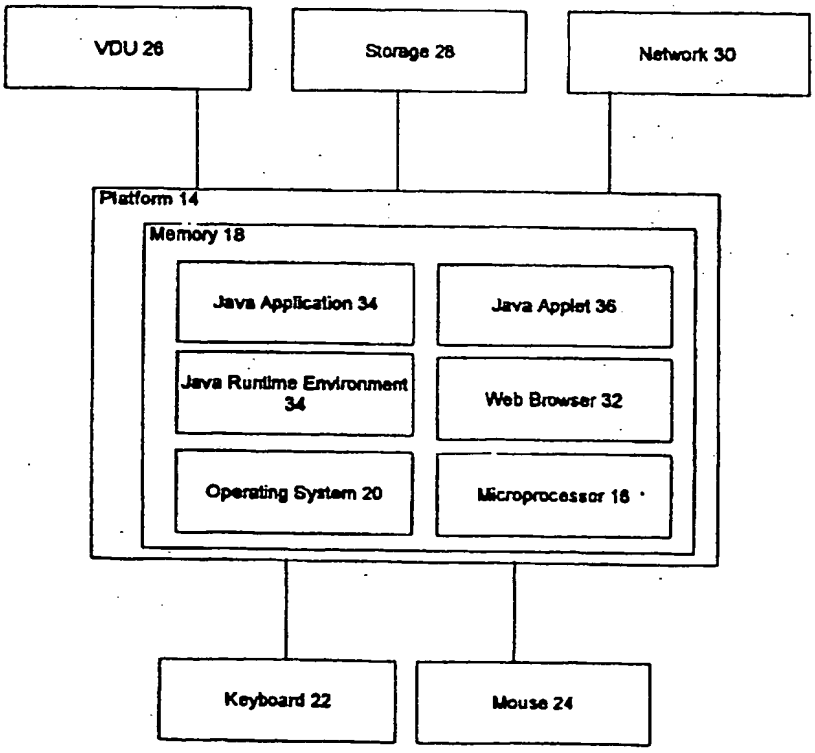

Figure 1

GB 2 343 022 A

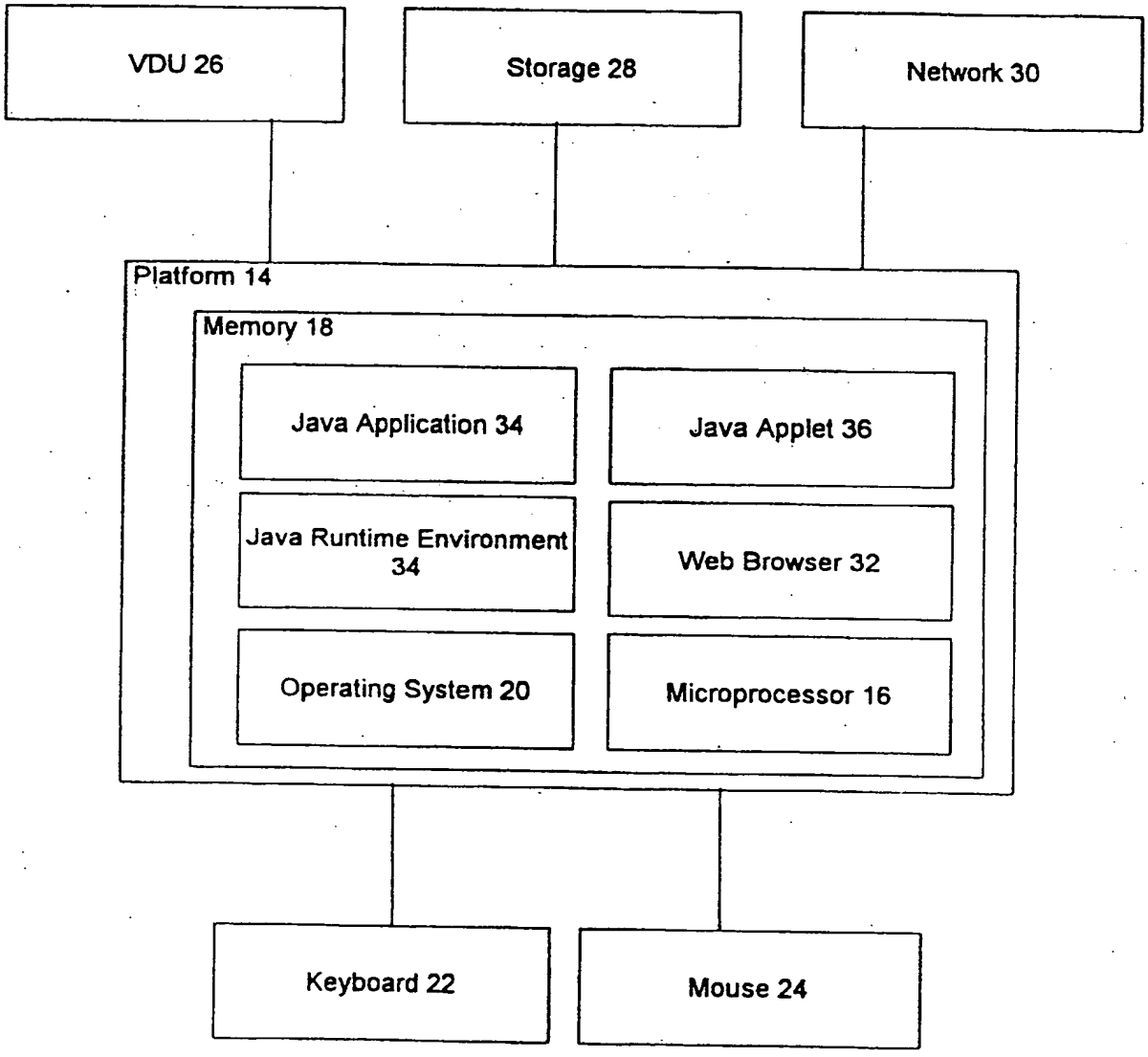
VDU 26
Storage 28
Network 30
Platform 14
Memory 18
Java Application 34
Java Applet 36
Java Runtime Environment 34
Web Browser 32
Operating System 20
Microprocessor 16
Keyboard 22
Mouse 24

Figure 1

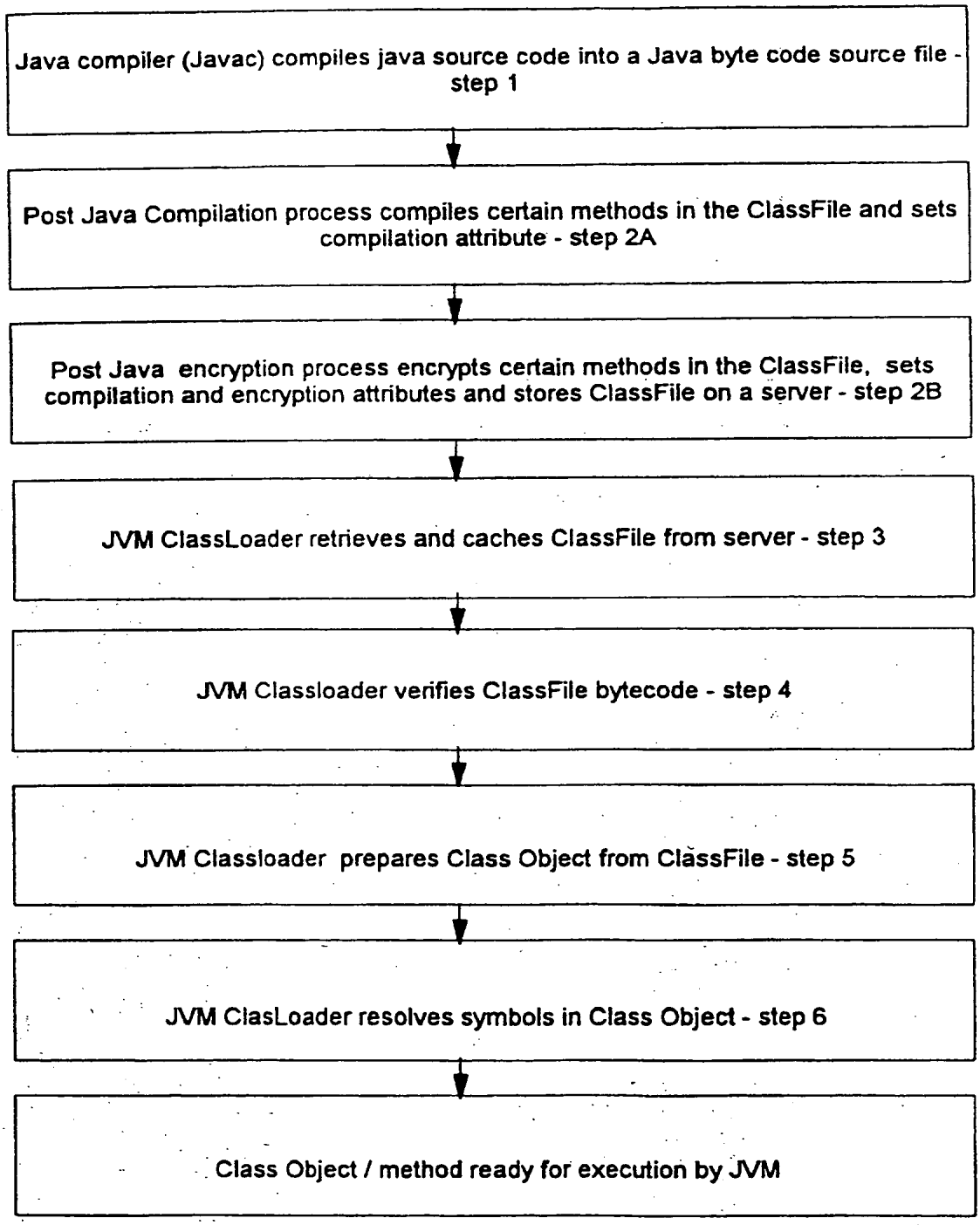
Java compiler (Javac) compiles java source code into a Java byte code source file - step 1
Post Java Compilation process compiles certain methods in the ClassFile and sets compilation attribute - step 2A
Post Java encryption process encrypts certain methods in the ClassFile, sets compilation and encryption attributes and stores ClassFile on a server - step 2B
JVM ClassLoader retrieves and caches ClassFile from server - step 3
JVM Classloader verifies ClassFile bytecode - step 4
JVM Classloader prepares Class Object from ClassFile - step 5
JVM ClasLoader resolves symbols in Class Object - step 6
Class Object / method ready for execution by JVM

Figure 2

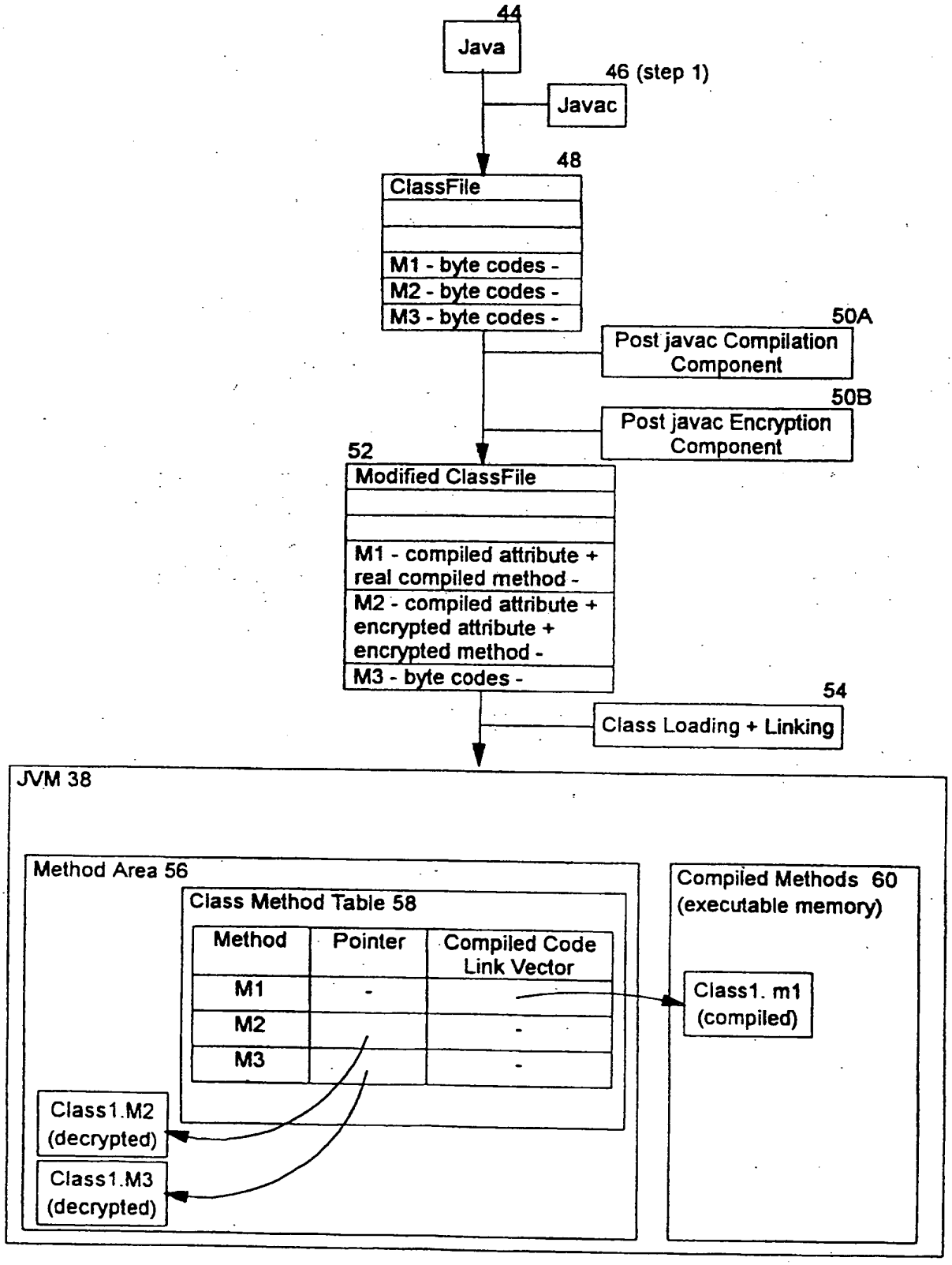
44
Java
46 (step 1)
Javac
48
ClassFile
M1 - byte codes -
M2 - byte codes -
M3 - byte codes -
50A
Post javac Compilation Component
50B
Post javac Encryption Component
52
Modified ClassFile
M1 - compiled attribute + real compiled method -
M2 - compiled attribute + encrypted attribute + encrypted method -
M3 - byte codes -
54
Class Loading + Linking
JVM 38
Method Area 56
Class Method Table 58
Method
Pointer
Compiled Code Link Vector
M1
M2
M3
Class1.M2 (decrypted)
Class1.M3 (decrypted)
Compiled Methods 60 (executable memory)
Class1. m1 (compiled)

Figure 3

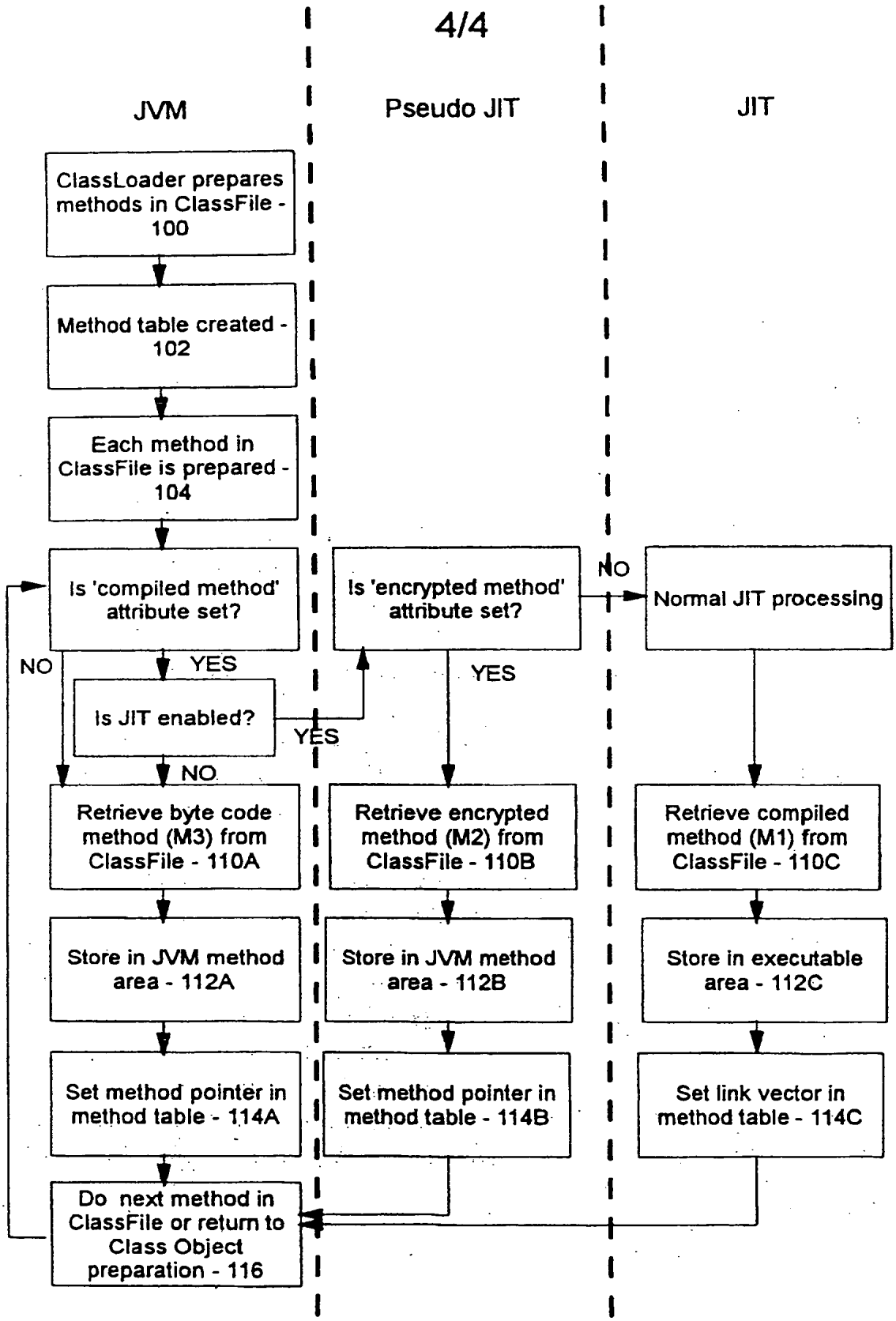
JVM
Pseudo JIT
JIT
ClassLoader prepares methods in ClassFile - 100
Method table created - 102
Each method in ClassFile is prepared - 104
Is 'compiled method' attribute set?
NO
YES
Is JIT enabled?
YES
NO
Is 'encrypted method' attribute set?
NO
YES
Normal JIT processing
Retrieve byte code method (M3) from ClassFile - 110A
Retrieve encrypted method (M2) from ClassFile - 110B
Retrieve compiled method (M1) from ClassFile - 110C
Store in JVM method area - 112A
Store in JVM method area - 112B
Store in executable area - 112C
Set method pointer in method table - 114A
Set method pointer in method table - 114B
Set link vector in method table - 114C
Do next method in ClassFile or return to Class Object preparation - 116

Figure 4

ENCRYPTING OF JAVA METHODS

This invention relates to the encryption of Java methods and in particular to the use of encryption as a means of concealing the actual implementation of a distributed Java program and to limit its use.

BACKGROUND

One of the advantages of the Java language is the high level nature of the byte code, this allow a programming language which is simple to understand and platform independent. A disadvantage of high level byte is that it may be converted into easy to understand source code. For instance, byte code retains the label and variable names which are readable on recompilation. Software developers are discouraged from developing Java applications due to the ease with which Java classes can be reverse engineered. Much proprietary information in Java application can be revealed through reverse engineering.

A solution to the problem of reverse engineering to use encryption to obscure the transmitted form of the class to make Java programs at least as secure as conventional, native object code. However, the application must be decrypted before it can be used which limits the usage of the application to users who know how to decrypt.

Another solution is to encrypt the application before shipping, ship the encrypted application and decrypt before installing on the machine. This solution works for stand alone applications which have their own installation programs to integrate them with their host machine. However applets on the internet are downloaded directly by a JVM and do not have any preprocessing that may decrypt them. The JVM is defined by Sun Microsystems and controls the downloading of applets. There is no present plans to modify the JVM to deal with encrypted applets and any unilateral modification would render that particular JVM non-standard. The difficulties involved in modifying such a pervasive and standard component for one particular type of decryption render this approach inappropriate.

The Java Virtual Machine (JVM) is a key component the Java programming language. It is particular for the operating system and the platform, it interprets Java byte code instructions from a Java application and uses built in routines and operating system routines to

process them. One of the disadvantages of interpretation of byte code by the JVM is that the same byte code may be interpreted many times. This is one reason why Java interpretation is considerably slower to process than compiled code (for instance a C++ object code program).

Compiling the Java byte code, instead of interpreting it, increases the speed of execution as the platform dependent code is then executed directly. The platform specific code is stored by the Just-in-Time (JIT) and used whenever the byte code would have been interpreted and used. The JIT is a platform-specific software compiler often contained within VMS. It compiles Java byte codes on-the-fly into native machine instructions, thereby reducing the need for interpretation.

However JIT compilation takes time to perform, if the byte code is only used once then it may be a false optimisation to compile it first - simple interpretation would in fact be better. By monitoring the byte code usage and only compiling that which is used most an increase in speed can be attained. This approach is being implemented in future releases of the JVM. The developer may also pre-compile those parts of the Java byte code that would benefit most from compilation. Platform dependent code for several machines could be included with the platform independent code so that the Java application would be optimised for the chosen platforms and still work (but not optimised) on those platforms that were not chosen. Obviously this will not benefit all platforms which diminishes the main advantage of Java but a significant advantage will be achieved for the most used platforms.

Java applications are made up of a number of Java ClassFiles. Each ClassFile represents a Java object oriented class and comprises fields containing the data necessary to build a class object in the JVM. The fields include fields to store the class properties, class methods for the execution of the class and attributes used by the JVM to link the class. A ClassFile is retrieved by the JVM's ClassLoader so that it may be verified for syntax, linked into the JVM and have its references resolved. Among the attributes is a 'compiled method' attribute which indicates to the JVM that the ClassFile includes platform dependent machine code for a particular method, the ClassFile may also contain the Java byte code for that method.

When the JVM retrieves the ClassFile it reads the attributes, if a method is not indicated as having compiled method code attached then the

method byte code is placed into the JVM method area memory. If a method is indicated as having compiled code attached then the JVM passes responsibility of the compiled code to the Just-In-Time compiler. This is because it is the JIT which handles the storing of compiled code in execution memory and the compiled code handling must be consistent for both precompiled code and JIT compiled code.

SUMMARY OF THE INVENTION

According to one aspect of the present invention there is provided a method as described in claim 1.

By making the ideas in software more secure, companies will be more likely to develop serious Java applications. By using the existing JIT mechanisms, such protection is not dependent upon specialised JVM technology at the client and an application vendor must supply a suitable JIT implementation. This allows an embodiment of the invention to be incorporated into any existing JVM. The level of security is open; it can range from a hard-coded key to on-the-fly encryption using a once only key delivered during the decryption process. In the latter case, the distributed class is only able to be decrypted on the occasion it is downloaded.

BRIEF DESCRIPTION OF DRAWINGS

In order to promote a fuller understanding of this and other aspects of the present invention, an embodiment will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a schematic representation of a computer system including a Java environment;

Figure 2 is a flow diagram of the processing steps of turning Java source code into Java objects for execution in the JVM;

Figure 3 is a schematic representation of the transformation of Java source code into Java objects linked to the JVM; and

Figure 4 is a flow diagram showing the processing of a Java ClassFile during linking by a JVM ClassLoader.

A Java environment 10 resides in a powered up operational computing system 12 such as shown in Figure 1. The computing system comprises, for

example, a computer platform 14 having Pentium II based microprocessor 16, 64k RAM memory 18 and Microsoft Windows NT operating system 20. Connected to the platform 14 are computer peripherals: keyboard 22, mouse 24, VDU 26 and storage systems 28 such as a hard drive. The platform 14 is also connected to a network 30 via a network adapter and through the network to the Internet via an Internet gateway. When powered up the computer system first loads the operating system 20 into the memory from the storage system 28. The operating system 20 typically controls the loading of the Java environment 10, browser software 32 such as Netscape's Navigator and other software applications from the storage system 28. The operating system 20 typically controls the memory allocation of the computing system 12 and makes space in the memory for storing Java applications 34 and web pages with embedded Java applets 36. Java applications may be loaded from the storage system or downloaded from the internet using the browser 32 which is connected to the world wide web. The browser 32 may also download web pages with embedded Java applets. The main components of the Java environment 10 in the present embodiment are a Java Virtual Machine 38 (JVM); a Just In Time Java Compiler 40 (JIT) and a pseudo Just-In-Time compiler 42. The JVM 38 and the JIT 40 are part of the Java Development Kit and may be written by different companies. For instance Sun Microsystems JDK 1.1 for Windows contains a Sun JVM and a Symantec Corporation JIT. In the present embodiment it is anticipated that the same provider would supply the JIT and the Pseudo JIT to the Java environment.

In Figure 2 the processing of Java source code 44 (Figure 3) into Java objects for execution in the JVM 38 (Figure 1) is shown. Representations of the Java code in Figure 3 correspond with some of the steps in Figure 2. A Java compiler 46 (Javac) retrieves Java source code 44 as written by a developer using an editor and converts it into the machine independent byte code which is recognised by a JVM running on any platform (step 1). The byte code is a collection of object orientated classes, each class being contained in a ClassFile 48 (Figure 3) . The ClassFile 48 shows the byte code for three methods M1,M2,M3 encapsulated within the ClassFile. A post Javac compilation process (step 2A) is applied by a post Javac compilation component 50A to the ClassFile 48 which compiles certain methods (of the developers choosing) and sets a 'compiled method' attribute in the ClassFile. Afer post Javac processing the ClassFile 48 becomes a modified ClassFile 52. The compiled method is represented in Figure 3 by method M1 in the Modified ClassFile 52. In the embodiment a post Javac encryption process (step 2B) is applied by a post

Javac encryption component 50B to the ClassFile 48 to encrypt certain methods, in this example method M2. The 'compiled method' attribute is set. Also a newly defined attribute (this is allowed in the Java Language Specification) - an 'encrypted method' is set, this is represented by method M2 in Figure 3. Method M3 is not post Javac processed and remains pure Java byte code as indicated in Figure 3. The modified ClassFile 52 is stored with other ClassFiles as part of a Java application or Java applet. The Java application may be on a server or on the hard drive of a connected client, the Modified ClassFile is transferred to the computer system when the application is loaded from the server or the hardrive. A Java applet is part of a web page and the Modified ClassFile 52 is transferred to the computer system 12 when the web page is downloaded by the browser 32.

The JVM 38 comprises a ClassLoader 54 (Figures 3 and 4) which retrieves the Modified ClassFile 52 from the server or from storage and caches it (step 3) in preparation for integration with the JVM 38. The JVM ClassLoader 54 verifies the byte code (step 4) by checking the syntax against the Java Language specification, if any errors in the byte code are found they are returned and the loading stopped. Next the JVM Classloader 54 prepares a class object from the Modified ClassFile 52 (step 5) , this integrates the class object with the JVM and allows the execution of the class object as part of the Java application or applet. The preparation or integration is also know as linking and involves, amongst many other things creating memory space for class object variables in a JVM heap, placing method byte code M2, M3 in method area memory 56 and dealing with compiled methods. That part of the class object preparation process used for compiled and encrypted methods which is part of the embodiment of the invention is shown in Figure 4. After preparation of the class object the JVM Classloader resolves any symbols into the class object (step 7), this involves retrieving any other ClassFiles that are referenced. The Class object is now ready for interpretation. The last steps (steps 4, 5, 6 of Figure 2) performed by the ClassLoader are not necessarily performed in the order given as this depends very much on the type of JVM used.

The Class Object preparation with respect to 'compiled method' code is represented in Figure 4. The Classloader prepares methods from the Modified ClassFile 52 for integration within the JVM (step 100). First a method table 58 is created in the method area memory 56. This table is used to reference the methods in the JVM. Three fields in the table are

described: the method name; the method pointer; and the compiled link vector for the method. The method name is a set of symbols representing the method, in this example the three method names are M1, M2 and M3. The method name is used as an index into the table to look up a pointer of the method byte code or a link vector for the compiled platform specific code. The pointer comprises a memory address where the byte code of the method resides. The link vector comprises a memory address in executable memory where the compiled platform dependent code resides, in this way the link vector is really a pointer as well. The JVM goes through each method in the ClassFile (step 104). For each method the JVM 38 checks whether the 'compiled method' attribute is set. If so then the JVM 38 normally passes control over to the JIT 40, however in this embodiment a 'compiled method' pointer which normally points to the JIT 40 points to the pseudo JIT 42 and control passes to the pseudo JIT 42 instead. If the 'compiled method' attribute is not set then the JVM continues and does not give up control, it retrieves the byte code method (in this example M3) from the ClassFile 52 (step 110A) and stores it in the JVM Method area 56 (step 112A). Then the pointer in the method table 58 is set at the address of the method M3 in memory (step 114A). The next method in the ClassFile is processed (step 116) by jumping back to step 106 and checking the 'compiled method' attribute of the next method. If no other methods remain then the JVM continues with the normal class object preparation.

If the 'compiled method' attribute is set in step 106c then control passes to the pseudo JIT 42. The pseudo JIT 42 checks to see if the 'encryption attribute' is set in the ClassFile (step 120) , if it is not set then control passes to the JIT 40 and processing of 'normal' compiled methods continues, if set then the encrypted method (encrypted by the post Javac encryption process in step 2B) is retrieved (step 110B). In this example the retrieved encrypted method is M2. The pseudo JIT applies a decryption algorithm and decrypts the method (step 111) and then stores the decrypted method in the JVM method area memory (step 112B). The pointer in the method table is updated with the method's location in the method memory area (step 114B) and control is passed back to the JVM to process the next method in the ClassFile (step 116).

If the 'encryption' attribute is not set in step 120 then control passes to the JIT 40. Normal JIT processing of a compiled method is carried out (step 120) including retrieving the compiled method (step 110C), storing the compiled method in executable memory (step 112C) and

setting the link vector in the method table (step 114C). Control is passed back to the JVM to process the next method in the ClassFile (step 116).

If the 'encryption' attribute is not set in step 120 then control passes to the JIT 40. Normal JIT processing of a compiled method is carried out (step 120) including retrieving the compiled method (step 110C), storing the compiled method in executable memory (step 112C) and setting the link vector in the method table (step 114C). Control is passed back to the JVM to process the next method in the ClassFile (step 116).

In summary there is described a method of processing a Java ClassFile component on a client comprising retrieving the ClassFile component from a server and checking the component for a compiled part which is indicated by a flag or attribute in the ClassFile. If a compiled part is detected further checking of the ClassFile is performed to see if the part is encrypted, again indicated by a flag or attribute. If the part is so encrypted it is then decrypted. Another aspect of the invention is a method of distributing software components which comprises encrypting at least a part of the software component and setting a 'compiled' attribute and an 'encrypted' attribute for each encrypted part of the component. The component is stored on a server where it might be accessed by a client.

CLAIMS

1. A method of processing a software component on a client comprising the steps of:

retrieving the component from a server;

checking the component for a compiled part;

checking, if a compiled part is detected, that said part is encrypted;

decrypting said part if so encrypted

2. A method as claimed in claim 1 wherein the software component is a Java ClassFile.

3. A method as claimed in claim 2 wherein the component is retrieved and checked for compiled parts by a Java Virtual Machine.

4. A method as claimed in claim 3 wherein the JVM passes control to pseudo JIT code if a compiled part is detected.

5. A method as claimed in claim 4 wherein said part is checked for encryption and decrypted if necessary by the pseudo JIT.

6. A method as claimed in claim 5 wherein the pseudo JIT passes control to a real JIT if said part is not encrypted and compiled.

7. A method of distributing software components comprising:

encrypting at least a part of the software component;

setting a 'compiled' attribute and an 'encrypted ' attribute for each encrypted part of the component; and

storing the component on a server where it might be accessed by a client.

8. A method as claimed in claim 7 wherein the software component represents a object oriented class and one or more of the methods of the class are encrypted.

9. A method as claimed in claim 7 wherein the software components are Java ClassFiles.

10. Apparatus for processing a software component on a client comprising:

means for retrieving the component from a server;

means for checking the component for a compiled part;

means for checking, if a compiled part is detected, that said part is encrypted; and

means for decrypting said part if so encrypted.

11. Apparatus as claimed in claim 10 wherein the software component is a Java ClassFile.

12. Apparatus as claimed in claim 10 or 11 wherein means for retrieving, means for checking the component for a compiled part are part of a Java Virtual Machine.

13. Apparatus for distributing software components comprising:

means for encrypting at least a part of the software component;

means for setting a 'compiled' attribute and an 'encrypted' attribute for each encrypted part of the component; and

means for storing the component on a server where it might be accessed by a client.

14. Apparatus as claimed in claim 13 wherein the software component represents an object oriented class and one or more of the methods of the class are encrypted.

15. Apparatus as claimed in claim 13 wherein the software components are Java ClassFiles.

16. A computer program product stored on a computer readable storage medium for, when executed on a computer, executing a data processing method of processing a software component on a client comprising the steps of:

retrieving the component from a server;

checking the component for a compiled part;

checking, if a compiled part is detected, that said part is encrypted;

decrypting said part if so encrypted.

17. A computer program product stored on a computer readable storage medium for, when executed on a computer, executing a data processing method of distributing software components comprising:

encrypting at least a part of the software component;

setting a 'compiled' attribute and an 'encrypted' attribute for each encrypted part of the component; and

storing the component on a server where it might be accessed by a client.